NASA/WVU Software IV & V Facility
Software Research Laboratory
Technical Report Series

NASA-IVV-96-011
WVU-SRL-96-011
WVU-SCS-TR-96-20
CERC-TR-TM-96-011

Design and Implementation of Replicated Object Layer

by Sudhir Koka

National Aeronautics and Space Administration

West Virginia University

According to the terms of Cooperative Agreement #NCCW-0040, the following approval is granted for distribution of this technical report outside the NASA/WVU Software Research Laboratory

George J. Sabolish Date
Manager, Software Engineering

John R. Callahan Date
WVU Principal Investigator

Design and Implementation of Replicated Object Layer

THESIS

Submitted to the Eberly College of Arts and Sciences
of
West Virginia University
In Partial Fulfillment of the Requirements for
The Degree of Master of Science

by
Sudhir Koka

Morgantown
West Virginia
May1996

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my advisor Dr. Jack Callahan for directing my research and helping me throughout my work here. I would also like to thank Dr. James D. Mooney and Dr. V. Jagannathan for agreeing to be on my Examining Committee.

An immeasurable amount of appreciation and gratitude go to Brian Whetten of USC Berkeley and Todd Montogomery for providing enormous expertise, knowledge and support throughout this work. I would like to thank people in NASA/WVU Research Lab and Concurrent Engineering Research Center for their support and cooperation.

Lastly, but most importantly, I would like to thank my parents and brother for unquestionable support, love and encouragement. Without them this would not have been possible.

Abstract

One of the widely used techniques for construction of fault tolerant applications is the replication of resources so that if one copy fails sufficient copies may still remain operational to allow the application to continue to function. This thesis involves the design and implementation of an object oriented framework for replicating data on multiple sites and across different platforms. Our approach, called the Replicated Object Layer (ROL) provides a mechanism for consistent replication of data over dynamic networks.

ROL uses the Reliable Multicast Protocol (RMP) as a communication protocol that provides for reliable delivery, serialization and fault tolerance. Besides providing type registration, this layer facilitates distributed atomic transactions on replicated data. A novel algorithm called the RMP Commit Protocol, which commits transactions efficiently in reliable multicast environment is presented. ROL provides recovery procedures to ensure that site and communication failures donot corrupt persistent data, and make the system fault tolerant to network partitions. ROL will facilitate building distributed fault tolerant applications by performing the burdensome details of replica consistency operations, and making it completely transparent to the application.Replicated databases are a major class of applications which could be built on top of ROL.

Chapter 1 Introduction

There has been a significant increase in the use of distributed systems over the last few years due to increasing availability of cheaper and faster computers, and new networking capabilities like multicasting. Reliability and availability of loosely-coupled distributed systems is a major requirement for distributed fault tolerant applications. One of the methods of making a distributed application tolerant to site failures and communication link failures is replication of resources, but a replica consistency protocol is required to maintain the consistency between the replicated data at different sites. Distributed fault tolerant applications need an approach that provides a mechanism for consistent replication of data over dynamic networks.

Replicated Object Layer (ROL) is one such approach, that provides an object-oriented framework for replicating data at multiple sites and across different platforms. ROL provides type registration facility to give structure to replicated data. Sun's External Data Representation [Sun87] is used to provide ship the data across different platforms. ROL provides support for performing distributed atomic transactions on replicated data. A novel algorithm called the RMP Commit Protocol, which commits transactions efficiently in reliable multicast environment is presented. ROL provides recovery procedures to ensure that site and communication failures donot corrupt persistent data, and make the system

fault tolerant to network partitions. Together, the RMP Commit Protocol and the recovery procedures provide a mechanism for consistent replication of data over dynamic networks.

A reliable communication protocol is usually required to maintain the consistency of state between functioning replicas and to mask replica failures. The Reliable Multicast Protocol (RMP) is the communication protocol used by ROL. RMP provides reliable delivery, message ordering guarantees and fault tolerance on top of an unreliable IP Multicast.

ROL will facilitate building distributed fault tolerant applications by performing the burdensome details of replica consistency operations and making it completely transparent to the application. Replicated databases are a major class of applications which could be built on top of ROL.

This section explains how a simple replicated database can be built on ROL. The data types for the replicated data can be created using the type registration facility provided by ROL. For example, if we are building a replicated employee database, then an employee table in a database can be described as an unbounded list of employee objects. The employee objects contain the replicated data of employee structure type created by ROL registration facility. The employee database is replicated on multiple sites belonging to a group called server group. ROL provides membership algorithms that allow sites to join or leave the group. Support for remote clients is provided to access the employee objects in the server group. Simple updates can be made to employee objects using read and write operations. The changes can be made on the fields of the employee object like name, salary etc. The updates made to the objects are totally ordered, that is all the sites in the group see the same sequence of updates. For more complex updates, distributed atomic transactions provided by ROL can be used. A client can begin a transaction and associate object updates to the transaction. The transaction has to acquire locks on the field of an object before making an

update to that field. When the transaction is committed the updates are made permanent and the locks held by the transaction are released. ROL provides support for fault tolerance to multiple site failures. The consistency of the replicated data is maintained. ROL provides all the functionality transparently to the application.

1.1 Preview of Chapters

This thesis is organized as follows: Chapter 2 gives a brief introduction to IP multicasting and RMP; discusses concurrency and recovery control concepts in distributed databases. Chapter 3 describes the design of ROL; presents a novel algorithm for committing transactions in reliable multicast environment; presents a finite state machine protocol which provides for membership changes and recovery of the group from site failures. Chapter 4 describes the implementation details of ROL and presents a sample application built on ROL. Chapter 5 discusses the approach taken for verifying state specifications of ROL. Chapter 6 presents the conclusions of the thesis.

Chapter 2 Related Work

This chapter presents background to IP Multicasting, group communication concepts and RMP. It then discusses the different approaches to replicating data and finally the concepts of concurrency control and recovery in distributed database systems are presented.

2.1 IP Multicasting

Multicasting is a technique used to pass copies of a single packet to a subset of all possible destinations. This technique has been supported on most local area networks. On these networks, a multicast packet to all of the hosts has the same overhead as a unicast packet to just one of them.

Internet Protocol supports multicasting [Deer89]. IP Multicast addresses are provided for a host to receive multicast traffic destined for a certain multicast group. IP Multicast employs best effort delivery and does not provide message ordering. Multicast routing across LANs is done using one of the routing protocols, Distance Vector Multicast Routing Protocol (DVMRP) [WPD88] or Multicast Open Shortest Path First (MOSPF) routing protocol [Moy94]. Since many of the present IP routers do not support multicasting, an ad hoc solution called *tunneling* is used to provide routing of multicast traffic. Tunneling is a scheme

that involves encapsulating an IP multicast packet inside regular IP unicast packet and sending the packet to another multicast capable router.

The multicast traffic on the Internet is controlled by pruning the routing tree to adaptively restrict traffic and by limiting multicast packet lifetime. Each multicast packet has a Time-To-Live(TTL) value. Only packets with a TTL value greater than 1 are able to leave the local network or subnet in which they were created. The multicast router decrements the TTL of a packet as it is transmitted across. Thus, packets with higher TTL have much larger scope than smaller TTL packets because the TTL can stay larger than most threshold values as packet travels among subnets.

The virtual network of interconnected multicast routers is called Internet *Multicast Back-Bone* (MBone). The MBone allows a single stream of information to be received by a large number of hosts that are distributed globally. The MBone is primarily used to broadcast live video and audio meetings, but recently several other applications like Computer Supported Cooperative Work (CSCW) applications and fault tolerant distributed applications have been developed using the MBone [Hans94].

2.2 Group Communication Concepts

Process and object groups, ordering and reliability of messages, and fault tolerance are some of useful concepts associated with development of distributed applications for group communication. A *process group* is a set of processes that interact to perform a distributed operation.Two frequently used group communication models are *publisher-subscriber* and *client-server* models. The publisher-subscriber model allows processes to join or subscribe to a group. Individual processes in the group may send messages for the other members in the group to receive. This model does not require explicit naming of message destinations.

plication and proceed. Together these concepts of resiliency and fault tolerance attempt to provide a system model that is robust in face of failures and has the ability to recover from failures transparently. RMP is a communication protocol which provides support for all the group communication requirements mentioned above.

2.3 Reliable Multicast Protocol (RMP)

RMP provides totally ordered, reliable and atomic multicast service on top of IP multicast[Mont94]. It provides a transport mechanism by which user can design and implement fully distributed, fault tolerant applications without need to deal with the low-level communication primitives. RMP supports the publisher-subsrciber and client-server communication models for the application developer.

2.3.1 RMP Entities

RMP is based on a model of complex interactions between processes operating on interconnected hosts. RMP is organized around three entities: *RMP Processes*, *Token Rings* and *Token Lists*. An RMP Process is a member of a process group that is using RMP for group communication. A group of processes communicating to achieve message ordering, is referred to as a *Token Ring*. Each process may be member of multiple token rings. Alternatively, processes may communicate with a token ring through the use of client-server communication model. The list of members of a token ring is called the *Token List*. Token lists are also referred to as *membership views* because they represent a view of the current membership of the token ring. RMP provides *locks* which are mutually exclusive. A member may hold one or more locks and the semantics attached to a lock are dependent on what the application requires the lock to do. Six of these locks are set aside as *handlers*. A handler

is a mechanism that allows a message to be sent to a group and have only one member reply or handle that message.

Processes that are not part of the token ring may send a message to the group, and optionally be either notified of its delivery to the group or receive a reply from one of the members. This model of RMP communication, referred to as *Multi-RPC*, allows RMP to support a client-server communication model. This allows processes that desire information from the token ring, but do not want to pay the overhead price of actually joining the Token Ring, to communicate directly with the ring.

2.3.2 Interaction Model

RMP is based on a modified version of family of protocols called Token Ring Protocols [CM84]. All the messages in the messages are serialized by the *token site*. The token site sends an ACK or acknowledgment, containing a special sequence number called a *timestamp*. This ACK is a multicast message sent in response to a data message sent to the group. In this way the system uses a positive acknowledgment scheme. Upon receipt of the acknowledgment, the message sender knows whether or not his message has been received by at least one receiver. The system also operates as a negative acknowledgment scheme to other receivers, who know if they miss a message because of the imposed timestamp ordering. Missing messages can then be requested by sending a NACK to the other receivers or the sender requesting a retransmission of the missing message. In addition, a message source places a sequence number on a message to order that message with respect to other messages it has or will send. Sequence numbers provide ordering of messages with respect to the same site. Timestamps, however provide global ordering across all sites. The infinite buffering of the negative acknowledgment is avoided by rotating the token as a conse-

quence of generating the ACK for a message and making it mandatory for a receiver to accept the token only if has all the timestamped messages. Livelocks are avoided by making token passes mandatory within a specified time interval.

2.3.3 Message Delivery and Fault Tolerance Features

The application is notified by RMP once token is transferred N times after a message has been acked, where N is the token ring size. This is referred to as *totally resilient* or *safe* delivery, which ensures that each receiver has the message before application sees it. If K sites accept the token after the message is timestamped, then K +1 sites would have to fail before the message is lost. This is concept of *k-resiliency* of messages. A message is referred to as *majority resilient* if more than half of the sites have the message. RMP allows application to select the desired *quality of service*(QOS) ranging from unreliable to totally resilient on per message basis. The message is delivered immediately to the application and RMP provides a RMP_MSG_QOSMET event notification once desired QOS is met for the message.

RMP provides efficient membership change extensions for a process to join and leave the ring. A flexible failure recovery model is provided by RMP which allows the system to return to normal operations in the face of specified minimum size partitions. The reformation protocol synchronizes the reformed sites to a common synch point, that is all sites have the same set of packets. If there is a message loss and the common synch point cannot be reached, then it is referred to as an *atomicity violation* and application is notified about it. RMP allows application to set fault tolerance level, which refers to the minimum number of members required for the partition to proceed, to be set on per Token Ring basis.

So far, the group communication concepts and a communication protocol that provides support for these concepts have been discussed. The next sections discuss about replicated data and replication approaches in distributed systems.

2.4 Replicated Data

A replicated data application is one in which multiple copies of some data items are stored at multiple sites. The main reason for using replicated data is to increase availability. By storing critical data at multiple sites, the application can operate even though some sites have failed. Another goal is improved performance. Since there are many copies of each data item, a transaction is more likely to find the data it needs close by, as compared to a single copy application. This benefit is mitigated by the need to update all copies of each data item. Thus, read operations may run faster at the expense of slower write operations.

2.5 Replication Approaches

Two main approaches for replication known in the literature are *primary-backup* and *active replication.*

2.5.1 Primary - Backup Replication

In the primary-backup approach, one of the replication servers, the primary receives, evaluates and responds to invocations from clients. To ensure that other servers stay mutually consistent the primary must send a checkpoint(snapshot) of its state to other members. Some primary-backup architectures allow backups to respond to queries in order to increase the system performance. If the primary crashes, one of the backups takes over from the failed primary and resumes execution from the most recent checkpoint. No further in-

vocations can be processed until the new primary has been elected, which could degrade the performance of the protocol. Since only one replica ever performs an operation, the replicas need not be deterministic in nature.

2.5.2 Active Replication

Active Replication, in contrast, is a symmetric approach where each of the replication servers is guaranteed to invoke the same set of actions in the same order. This approach requires the next database state to be determined by the current state and the next action. Other factors, such as the passage of time, have no bearing on the next state. When implementing an active replica group it is necessary to ensure that all replicas receive and process the same sequences of request. This imposes an overhead on all client and replicated server interactions. Some active replication architectures replicate only updates, while queries are locally replied to improve the performance of the protocol.

2.6 Distributed Transactions

This section describes the distributed transactions. Section 2.6.1 describes about the use of locking mechanism for concurrency control and one of the locking policies, the Two Phase Locking is explained. The atomic commit protocols needed to commit a distributed transaction are covered in detail in Section 2.6.2.

Transaction is an execution of a program that accesses the shared data items. Concurrency control and recovery[BHG 87] are used to ensure that the transactions execute atomically, meaning that

- Each transaction accesses shared data without interfering with other transactions.

- If a transaction terminates normally(Commit), then all effects are made permanent, otherwise(Abort) it has no effect at all.

Concurrency control allows two or more transactions to execute in interleaved fashion without interfering each other, and always maintaining the consistency of the database.

2.6.1 Two Phase Locking

Locking is a mechanism commonly used to solve the problem of synchronizing access to shared data. The idea behind locking is intuitively simple. Each data item has a lock associated with it. Before a transaction T_1 may access a data item, the scheduler first examines the associated lock. If no transaction holds the lock, then the scheduler obtains the lock on behalf of T_1. If another transaction T_2 does hold the lock, then T_1 has to wait till T_2 releases it. The scheduler thereby ensures that only one transaction can hold the lock at a time, so only one transaction can access the data item at a time.

Transactions access data items either for reading or for writing them and hence there are two types of locks, read and write locks. Concurrent reads are allowed since interference cannot occur, but write operation needs exclusive access to the data item. Two locks conflict if they are issued by different transactions, and one or both of them are write locks.

Two phase locking policy[EGLT76] is used to guarantee the serializabilty of transactions. Using such a scheme, no new locks can be acquired after any lock has been released, resulting in two distinct phases during the lifetime of a transaction, a growing phase where locks are being acquired, and a shrinking phase where locks are being released. This policy could result in deadlocks where before either of two transactions can proceed, one must release a lock that other needs to proceed. One of the strategies of preventing a deadlock forever is to use time-out and abort one of the transactions.

2.6.2 Atomic Commit Protocol

Atomic Commit Protocol(ACP) is an algorithm for the coordinator and participants such that either the coordinator and all participants commit the transaction or they all abort it. Each process may cast one of two votes: Yes or No, and can reach exactly one of two decisions: Commit or Abort. An ACP is an algorithm for processes to reach decisions such that:

AC1: All processes that reach a decision reach the same one.

AC2: A process cannot reverse its decision after it has reached one.

AC3: The Commit decision can only be reached if all processes voted Yes.

AC4: If there are no failures and all processes voted Yes, then the decision will be to Commit.

AC5: Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

2.6.2.1Two Phase Commit Protocol

The simplest and most popular ACP is the two phase commit (2PC) protocol[Gray78]. The simplest version of 2PC is centralized. One of the site is designated as the *coordinator*. The coordinator sends a request to prepare to commit to all the participants. Each site answers by a Yes (ready to commit) or by a No (abort) message. If any site votes No, all the sites abort. The coordinator collects all the responses and informs all the sites of the decision. In absence of failures this protocol preserves atomicity.

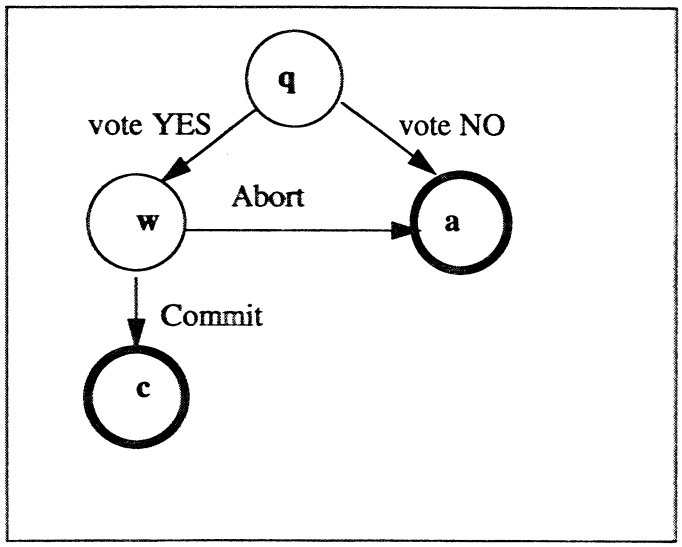

Figure 1 State Diagram for Two Phase Commit Protocol

Commit protocols can be described using state diagrams[SS83]. In Figure 1, the final states are circles with thick outlines. In this protocol, each site (either coordinator or slave) can be in one of four possible states. The state **q** is the *initial* state, a state in which a site remains till it decides whether to unilaterally abort or to agree to commit the transaction. In the wait state **w**, a coordinator waits for votes from all of the slaves, and each slave waits for the final word from the coordinator. This is the *uncertainty period* for each site as it does not know whether the transaction will be committed or not. The site transitions from wait state to commit state **c** if the decision to commit was made or else it transitions to abort state **a**. A site is in a *committable* state only if it knows that all the sites have agreed to proceed with the transaction. The rest of the states are *non-committable*. The only committable state in 2PC is the commit state.

Two phase commit is a blocking protocol. If the coordinator fails, all the sites may remain blocked indefinitely, unable to resolve the transaction. Locks must be held in the database while the transaction is blocked, thus rendering data inaccessible by other requests. It has been proved in literature [SS83] that there exists no non-blocking protocol resilient to network partitioning. When a partition occurs, the best protocols allow no more than one group of sites to continue while the remaining groups block. Skeen suggested the quorum based three phase commit protocol, that maintains consistency in spite of network partitions [Ske82]. This protocol is blocking, an operational site can be blocked until a failure is mended. In case of failures, the algorithm uses a quorum or majority based recovery procedure to resolve the transaction.

2.6.2.2Quorum based Three Phase Commit

The 3PC protocol is similar to two phase commit, but in order to achieve resilience, another non-final *buffer* state is added in 3PC, between wait and commit states. An intermediate state, *pre-commit* state **pc** is needed to allow for recovery. After collecting all the votes from sites, the coordinator sends a pre-commit if all responded with Yes votes else it aborts the transaction. Upon receiving the pre-commit the participant sends an ACK to coordinator. The coordinator sends a commit when it receives a quorum (majority) of ACKs else it blocks for more votes.

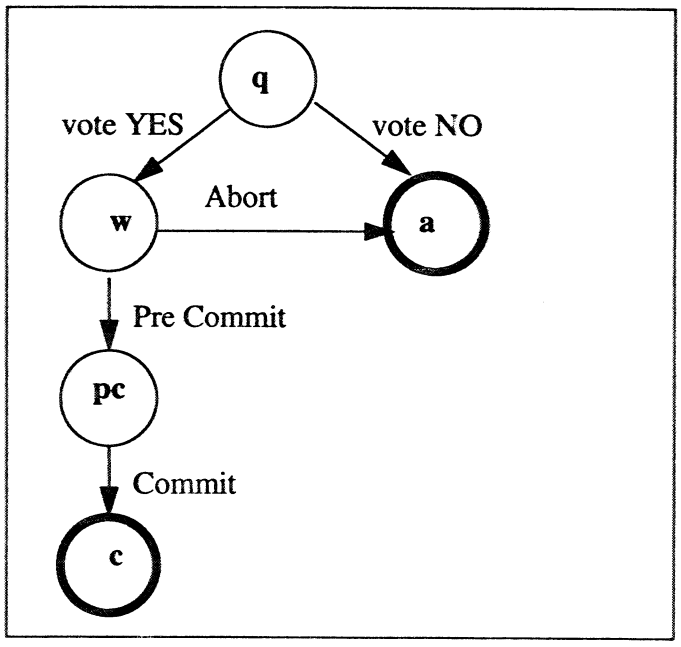

Figure 2 State Diagram of Three Phase Commit Protocol

The commit and pre-commit states are committable states in 3PC. In each step of the protocol, when the sites change their state, they must write new state to stable storage. When a group of sites detect a failure (a site crash or a network partition), or a failure repair (site recovery or merging of network partitions), they run the recovery procedure in order to try to resolve the transaction. The recovery procedure consists of two phases, first elect a new coordinator, and next attempt to form a quorum that can resolve the transaction. After collecting the states from all the sites, the coordinator tries to decide how to resolve the transaction. If any site has previously committed or aborted, then the transaction is immediately committed or aborted accordingly. Otherwise, the coordinator tries to establish a quorum. A commit is possible if at least one site is in the pre-commit state and the group of sites in the wait state together with the sites in pre-commit state form a quorum or a majority. Otherwise, the sites block till the network partitions are recovered.

Chapter 3 Design of ROL

This chapter discusses the overview of ROL and then describes the various guarantees provided by RMP to ROL. The RMP Commit Protocol is presented next. Finally, the state model of ROL is described which specifies the membership change, virtual token ring and recovery algorithms.

3.1 Overview of ROL

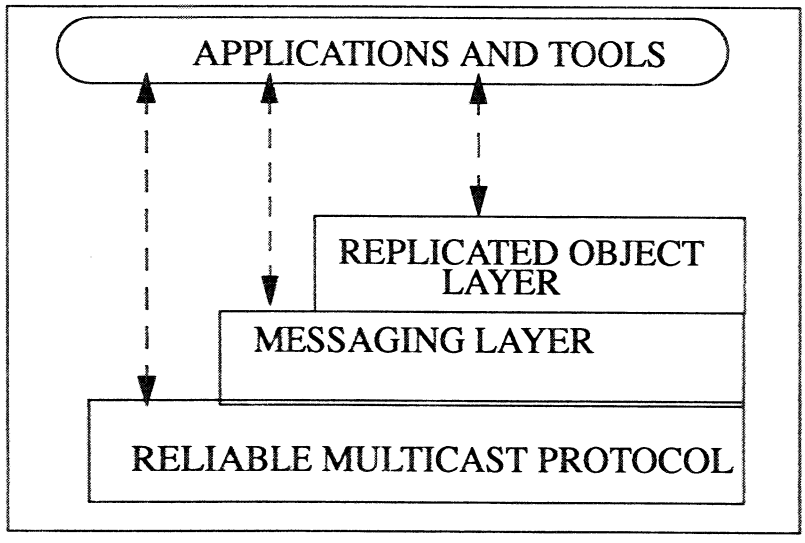

Figure 3 RMP Architecture

ROL exists as a component in Reliable Multicast Protocol Architecture. It is layered on top of Messaging layer. RMP provides reliable delivery, message ordering guarantees and fault tolerance on top of an unreliable IP Multicast. RMP handles data packets of sizes up to those that can be handled by the network. The messaging layer provides support for segmentation and reassembly of larger messages of sizes up to 5 Mbytes.

The ROL has two types of members *ROL Servers* and *ROL Clients*. The ROL Server is a member of a process group that uses RMP as its transport mechanism and protocols in ROL to maintain consistency of data. A ROL Server can be a member of multiple RMP

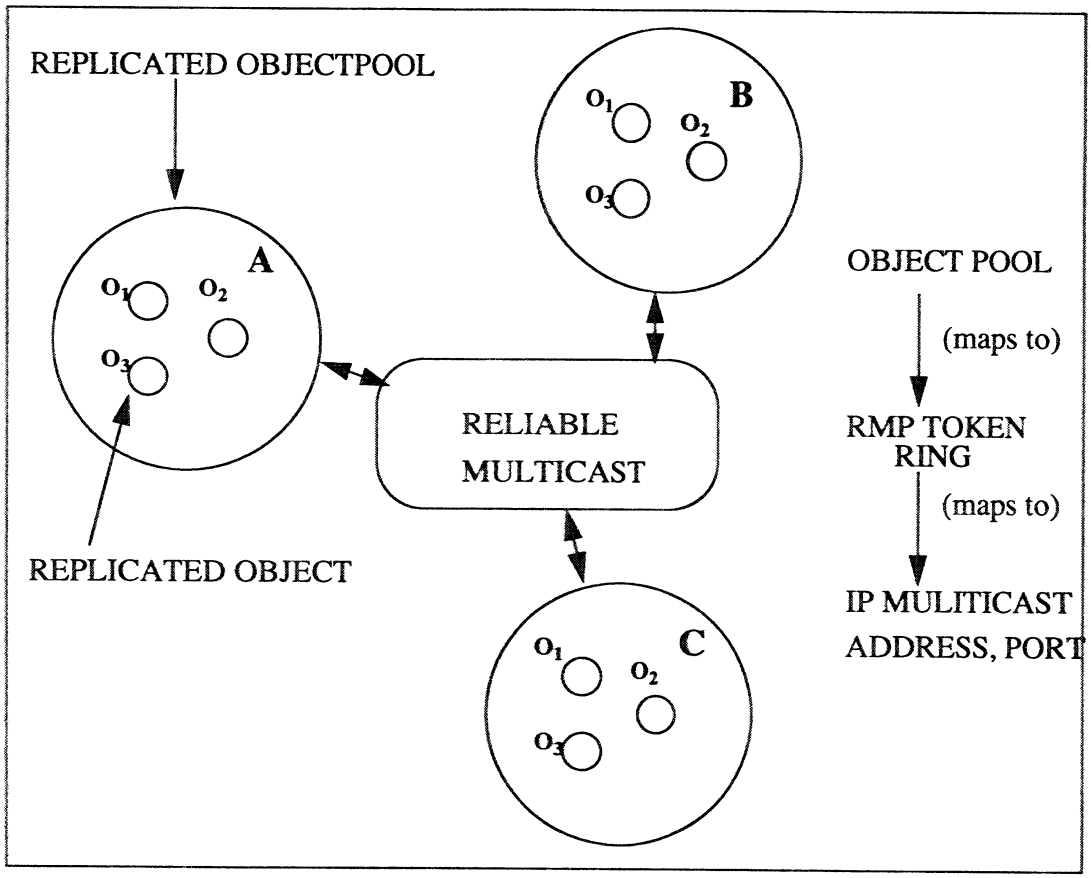

Figure 4 Replicated Object Pool

token rings. ROL Clients communicate with the ROL Servers using Multi-RPC.

ROL provides support for replicated objects. Each object has a type associated with it which can be registered by the type registration facility provided by ROL. Basic data types like integer, character etc. and composite types like structures and arrays can be created using this layer.

Objects are grouped into object pools, and each pool is replicated among all the members in the RMP token ring. The figure 4 shows ROL Servers A, B and C. For simplicity it is assumed in the figure that each server is a member of only one group and all servers belong to same group. The objectpool containing objects O_1, O_2, O_3 is replicated on these three ROL Servers in the group. An object group or pool in ROL maps to RMP Token ring in RMP layer, which in turn maps to a IP multicast address in IP layer. Thus, it is possible for a ROL Server to have multiple replicated objectpools each corresponding to the token ring it is a member of. ROL Clients can use Multi-RPC to get information about objects in a replicated pool.

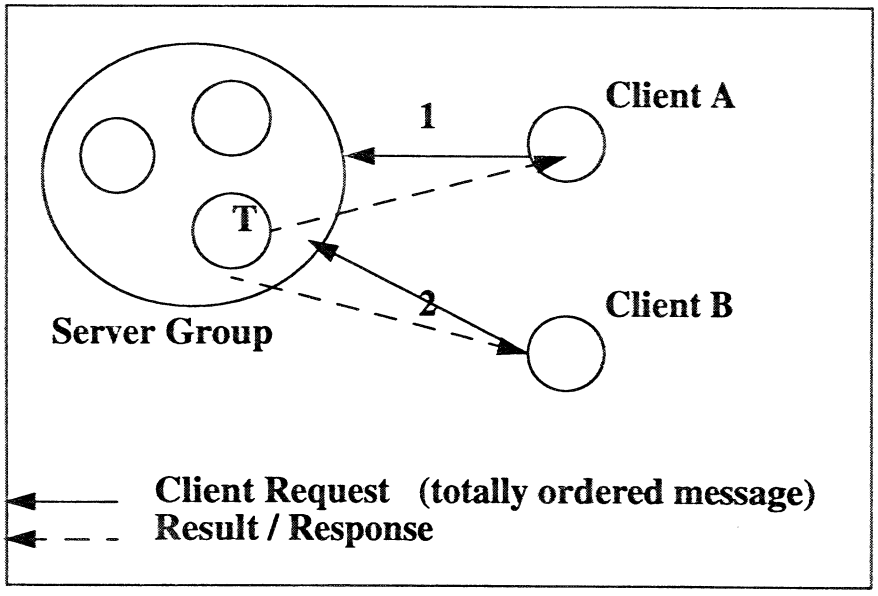

Figure 5 Active replication using totally ordered messages

ROL uses *active replication* approach to replicate data. All the sites in the group are replicas and they receive and process requests in same order. RMP provides reliability which means all the members in the group receive the request. RMP also provides ordering guarantees. The requests are sent to the replicas using totally ordered messages provided by RMP, thereby ensuring all the replicas receive the requests in the same order. The processing of the request must be deterministic, i.e, if all replicas are in identical objectpool state, then after processing a request all the replicas must be in the identical objectpool state. The ROL clients send requests to servers as totally ordered messages as shown in figure 5. One of the servers which is the token site sends back the result or response to the client.

ROL provides read and write operations to access and update objects respectively. Updates to objects are done with totally ordered messages, i.e all the members in the group receive the updates in the same order. Hence, the consistency of replicated data at different sites in a group is preserved in absence of majority partition failures. The updates are highly efficient, as RMP provides high throughput for totally ordered messages with low latency. ROL provides multi-granular read/write locks on replicated objects and facility to do transactions for more complicated updates. An ROL transaction uses two phase locking to guarantee serializabilty of operations and uses a special commit algorithm called *RMP Commit* to commit a transaction. While the transaction is going on, all the operations are written to local logs at all the ROL servers in the group. The transaction can either be committed or rolled back. All the locks acquired by the transaction are automatically released at the end of the transaction. One of the servers notifies the client about the result when the commit is done. A transaction is aborted if it ran more than a specified amount of time which helps in reducing deadlocks (refer to Section 2.6.1). ROL provides a membership change algorithm to allow ROL Servers to join and leave a group. To allow an objectpool to be replicated at multiple sites, this layer can copy objects to new sites, and can send updated copies

to sites which have partitioned or failed away and then rejoined. The layer also provides a recovery control mechanism if a majority partition fails.

3.2 RMP Guarantees and Features

The design of ROL is based on several guarantees and features that RMP provides [Mont94]. This section describes how ROL uses these features to provide a replicated object framework.

- Reliable delivery and total ordering of messages - The messages are received at all sites reliably in the same order. This property is used by ROL for replicating data consistently in the server group.

- Majority resilience of messages - The application is notified when the message has been delivered at majority of the sites in the ring. In the event of multiple site failures, the majority partition is guaranteed to have the message. This property forms the basis of RMP Commit Protocol.

- Virtual synchrony - The user can program as if the system is scheduled one distributed event at a time.

- Handlers - A handler is a mechanism that allows a message to be sent a group and have only one member reply or handle that message. This property is used in the virtual token ring algorithm.

- Reformation - The reformation protocol of RMP provides recovery from minority partition failures. The protocol synchronizes the packets at all sites and the state of Order-

ingQ is made identical at all sites. The reformation protocol is used by ROL to recover from minority partition failures in the ring.

- Fault tolerance levels - RMP allows application to set fault tolerance level, which refers to the minimum number of members required for the partition to proceed, to be set on per Token Ring basis. This property is used by RMP Commit Protocol to allow only majority partitions to continue.

The section 3.2.1 defines the term RMP Majority which forms the basis of RMP Commit Protocol and the section 3.2.2 describes the ROLO option provided by RMP for ROL to pass some information to all the members in the group on the back of RMP's control packets.

3.2.1 RMP Majority

Let's define RMP Majority as Size/2 + 1, where Size is the size of the group. The messages are received by at least RMP Majority number of members before the QOS is met. The fault tolerance level of the RMP ring is set to RMP Majority, i.e., if each partition after a failure must have at least RMP Majority number of members in it in order to continue operation, then at least one of them will always have any message eventually delivered to all members of the primary partition if one is able is to continue functioning.

3.2.2 ROLO option

RMP provides an ROL overload (ROLO) option to pass some information to all members with each token pass. The overload option allows the RMP token site to insert some data into the header of RMP control packets. The group members can read this information when they receive the control packet. The ACK and New List packets are the control pack-

ets that cause a token pass in RMP. RMP provides two callback functions, one to write some data into the control packet, and the other to read the data from the control packet. ROLO option is used by ROL to pass the status of a transaction at a member site to other members in the group.

3.3 RMP Commit Protocol

The coordinator-participant scheme of atomic commit protocols described in Section 2.6 becomes quite inefficient in RMP environment. The coordinator multicasts the prepare, pre-commits and commit messages to the group and the participants respond using unicast connections to coordinator. The three phase protocol commit incurs lot of messages and doesn't scale to RMP environment well. ROL uses the active replication approach i.e., ROL functions as a deterministic state machine where all the replicas in the group maintain the same state using totally ordered messages. Hence, RMP Commit protocol uses a decentralized protocol where all sites are equals instead of a coordinator-participant scheme.

Only a single commit message is multicast to the group to commit a transaction using this protocol. The status of transaction at different sites in the group is transmitted by piggybacking the status messages on the RMP ACKs using the ROLO option provided by RMP. The size of the status message is small compared to the size of the ACK packet, and hence does not hurt the performance of RMP. Thus, the transaction at each site proceeds depending upon the global information provided by the status messages " piggybacked " on ACKs. The RMP Commit protocol provides for pipelining of transactions, i.e., one transaction does not block the progress of another.

The commit protocol must handle two cases. The first case occurs if any of the sites continue processing, and the second is if all the sites stop functioning. For the sites which con-

tinue processing, either all of them have to receive the commit message or none of them will. If all sites stop functioning, at least P of the sites have flushed the commit message to disk so that it can be recovered. The persistence constant P, is a constant between 1 and RMP Majority - 1, which specifies the minimum number of sites that must have flushed a commit to disk before the commit is delivered.

In order to handle both these requirements, the transaction algorithm will not allow a commit message to be committed (or notified to the application) at any site until that site knows the Commit message has been flushed to stable store at a minimum of P sites, and has been received at the majority of sites in the group. When the first step is met, we say that it has been made *P-persistent*. The second requirement is met when the packet would be delivered with RMP's majority resilience. Each time that RMP gets an ACK or a NewList packet which does not have any holes or missing packets preceding it in the OrderingQ, RMP notifies the ROL of this condition using Token Pass notifications when the token pass meets total ordering requirements. The ROL can use this information to keep track of majority resilience.

ROL needs to know how many sites have flushed a commit message to stable store in order to decide whether that particular message has achieved P-persistence. Two problems were encountered while trying to determine the persistence of a commit message at a site. The first problem is that it will be a lot of overhead if the data packets are used to send this information. Instead RMP's ability to pass information with each token pass is used here. The RMP token site uses RMP's overload option to send the status of a flush. The second problem is that status of flushes of different Commit messages must be known, which means the Commit Message ID and the number of sites that have flushed that message has to be passed with each token pass. But this could add a significant amount of overhead to RMP's

performance. Hence, an assumption is made that flushes of all Commit messages are serialized according to the timestamp of the ACK that acknowledges them. Therefore the RMP token site sends the most recent timestamp that satisfies the condition that all the Commit messages with smaller timestamps have been flushed to disk on that site.

3.3.1 Data Structures for RMP Commit Protocol

Each site in the token ring has these data structures:

PersistenceTSP: All the Commit messages before and including this timestamp have been flushed to the disk at that particular site.

PersistenceTSL: List of the PersistenceTSP values at all the sites in the RMP token ring.

CommitQ: The ROL keeps track of the commit messages that are in progress through the use of CommitQ. The CommitQ holds a spot for each Commit message that the ROL knows about whose transaction has not yet been committed.

Each slot in the CommitQ goes through the following states, in order.

- WAIT — Hasn't achieved the required persistence and majority resilience.
- DELIVERED — Required persistence and majority resilience are met.
- COMMITED — All the changes made by the transaction are made permanent to stable store.

Suppose a transaction A is in progress. All the updates made by A are logged on in the memory and note that no changes have been made to the data A is operating on. The transaction A is committed by sending a commit message to all the sites. The QOS of message

delivery used for commit message is majority resilient. The commit message contains the ID of the transaction to be committed and the required persistence. When the commit message arrives, it is put into CommitQ. The site fills up each slot in the CommitQ with other information that is needed for the commit to proceed such as the timestamp of commit message packet and the process id's of sites participating in the transaction. The state of the slot is then set to the *wait* state. If there are no commit messages before this commit message to flush, then each site tries to flush contents of commitQ slot of the commit message and the updates made by the transaction to the stable store. After the flush is completed, the site updates the value of the PersistenceTSP to the timestamp of the commit message packet. The PersistenceTSP is then transmitted to the group using ROL overload option when the site becomes RMP Token Site. Other sites read the value of PersistenceTSP sent by the current RMP Token site and update the values in the PersistenceTSL for the token site. Periodically, the persistence of the transaction is checked, and if it has acquired the minimum persistence and the majority resilience (that is message is received by majority of the members), then the commitQ slot transitions to *delivered* state. The updates made by the transaction are made permanent in the memory and the application is *notified* about the successful completion of the transaction once slot transitions to this state. The locks held by the transaction are released at this point. Once the updates are made permanent to the stable store the slot transitions to committed state. If the site didn't flush the contents of commit packet to the disk in wait state, then it updates the value of PersistenceTSP. This helps in garbage collection of records flushed in wait state. When the persistence of the transaction equals the number of sites involved in the transaction, then these records are deleted. The slot is dequeued once the slot is in committed state.

3.3.2 Consistency of RMPCommit Protocol

The consistency of RMPCommit protocol is informally proved here by considering all the possible failure cases:

Case I: For all the sites continuing processing, either all of them commit the transaction or none of them will.

(a) If a minority of the sites fail and the transaction is committed, then one of the sites continuing processing must have the commit message since it has been committed after majority resilience is achieved. During reformation, RMP synchs the messages at all the sites to a common point and all the sites have the commit message.

(b) If a majority of sites fail, then the recovery algorithm given in section 3.4.2 is used.

Case II: If all sites fail, then a recovery algorithm given in section 3.4.2 is used.

The recovery algorithm described in section 3.4.2 recovers all the commits that have taken place in the ring, from majority or all site failures. The basic principle of the algorithm is that the commit message and the updates made by the transaction are flushed to P sites where P is between 1 and RMP Majority - 1. If the number of sites taking part in the recovery algorithm is N - P + 1 sites (from the old ring) then one of the sites must have information on all the commits, since flushing of all the commits to the disk is serialized.

3.3.3 Performance of RMPCommit Protocol

The performance of RMP Commit Protocol is dependent upon stable store processing and the time taken by RMP to pass the token in the ring. If the stable store processing is fast enough then the transaction should achieve minimum persistence by the time it achieves majority resiliency. The main advantage of the protocol is that it notifies the application immediately after the two conditions of minimum persistence and majority resilience are met. It does not have to wait for the transaction to be flushed to stable stores at all sites.

The optimistic latency is the time needed by the transaction to gain majority resilience and the transaction has achieved minimum persistence in that time. The time taken by a transaction to achieve majority resilience is (N/2 + 1) token passes after the commit message has been acked (i.e. N/2 + 1 members have the message). The actual latency is the time needed by the transaction to gain both majority resilience and minimum persistence.

$$\text{Optimistic latency} = T_{\text{majority resilience}} = (N/2 + 1) \text{ Token Passes}$$

$$\text{Actual latency} = \text{Max}(T_{\text{majority resilience}}, T_{\text{min. persistence}})$$

The tables 1 and 2 tabulate the theoretical performance figures for RMP Commit Protocol using the optimistic latency. It takes N/2 + 1 token passes to commit a transaction. A token pass involves a ACK and one or many DATA messages. Hence, at least N + 2 messages are needed to commit a transaction. Multiple commit messages can also be bundled into one message to improve the performance of the protocol. Another option provided by RMP is multiple data packets can be acked by a single ACK packet. N + n + 1 messages are need-

ed to commit n transactions at different sites, but whose commit messages are acknowledged by one ACK.

No. of Commits	Messages	No. of Messages
One Commit	(N/2 + 1)(ACK + DATA)	N + 2
n Commits acknowledged by one ACK	(N/2 + 1)(ACK + DATA) + (n - 1)DATA	N + n + 1

TABLE 1 Performance of RMP Commit Protocol without pipelining

The performance of the commit protocol can be improved by pipelining the transactions If the commit messages are acknowledged after one another, then only an additional ACK and DATA messages are needed to commit two transactions.

No. of Commits	Messages	No. of Messages
2 Commit messages acked one after another	(N/2 + 2)(ACK + DATA)	N + 4
m Commit messages acked one after another	(N/2 + m)(ACK + DATA)	N + 2m
m rounds of Commit messages acked one after another with n messages acknowledged by each ACK	(N/2 + m)(ACK + DATA) + m(n - 1)DATA	N + mn

TABLE 2 Performance of RMP Commit Protocol with pipelining

For example, a commit message is sent out as a totally ordered message and gets a timestamp of 9. The required Persistence is set to 2 and the size of the group is 9, so RMP Majority is 5. Assume that the sequence of ACKs and New List packets are ACK(9), ACK(10, 11), NewList, ACK(12), ACK(13), ACK(14, 15), NewList., ACK(16). The ACK(14, 15) satisfies the majority resilience for the commit message as five token passes have taken place and majority of the sites in the group have the message. Since there have been five

token passes, five sites have been the RMP token sites one after other, before the majority resilience has been achieved. A site can become RMP token site only if it has all contiguous timestamps and associated packets upto and including the token transfer that is making it token site. Hence all the five sites in the token ring after the token site which sent out ACK(9) have received the commit message. If the flushing of the commit message is fast enough given the speed of today's processors, there is a high probability that two of the five token sites have succeeded in their flushes and sent out their updated PersistenceTSP to the group. If the minimum persistence is met by the time the commit message gains majority resiliency, then the time taken to commit a message is only five token passes, which is the best case performance.

3.4 ROL State Specifications

The state representation of ROL is represented as state transition tables. These tables specify the membership change and recovery algorithms. It also describes the virtual token ring algorithm implemented in this layer to handle the load balancing among the replicas. The state representation is intended to facilitate the verification and validation of the ROL protocol. The state machine is driven by events which are reception of *RMP Events* [MCW95], *ROL Data* and *ROL Control* packets, and *expiring timers*. A state transition takes place on occurrence of one particular event and a condition being true. The condition applies before the actual state transition took place. The state representation is divided into two parts, normal and reformation modes. The reformation mode describes the recovery operations performed by ROL when the RMP reformation fails.

Event	Description
Data	Reception of multicast ROL Data packet
Unicast Data Packet	Reception of unicast ROL data packet
Token Pass	Reception of Token Pass packet (ROL Control Packet)
Update Pool Start	Reception of Update Pool Start packet (ROL Control Packet)
Update Pool End	Reception of Update Pool End packet (ROL Control Packet)
Synch Commits Request	Reception of Synch Commits Request packet (ROL Control Packet)
Abort Reformation	Reception of Abort Reformation packet (ROL Control Packet)
Resume Normal Operation	Reception of Resume Normal Operation packet (ROL Control Packet)
Unicast NACK packet	Reception of Unicast NACK packet (ROL Control Packet) NACKs for commit messages
Pass Token Alarm	Pass Token Alarm has expired.
Recv Token Alarm	Recv Token Alarm has expired.
Leaving Group Alarm	Leaving Group Alarm expired.
RMP_GRP_CHANGE	Reception of RMP_GRP_CHANGE, RMP Event which notifies about a RMP group membership change
RMP_GRP_REFORM	Reception of RMP_GRP_REFORM - RMP Event which notifies of successful reformation.
RMP_GRP_JOIN	Reception of RMP_GRP_JOIN - RMP Event which notifies on successfully joining a RMP group.
RMP_GRP_FORM	Reception of RMP_GRP_FORM - RMP Event which notifies about formation of RMP group.
RMP_GRP_RCVLCK	Reception of RMP_GRP_RCVLCK - RMP Event which notifies on receiving a handler/lock.
RMP_GRP_DENLCK	Reception of RMP_GRP_DENLCK - RMP Event which notifies on denial of a handler/lock.
RMP_FAILREF	Reception of RMP_FAILREF - RMP Event which notifies about failed reformation of a RMP group.
RMP_ATOMV	Reception of RMP_ATOMV - RMP Event which notifies about atomicity violation in the RMP group.

TABLE 3 Description of Events

3.4.1 Normal Operation of ROL

Each site in ROL can be in five possible states *Joining Group*, *Token Site*, *Not Token Site*, *Leaving Group* and *Not in Group*. Each site maintains a list data structure used by the state machine called the *Membership List*, describing the process id, position of the member in the RMP Token list and state of each member in the group. RMP Token list is identical at all sites and is used as a serialization mechanism between the ROL process groups. The membership is flushed to stable store each time it is changed. A *virtual token ring* is implemented in the ROL to do the load balancing among the replicas. The member in *Token Site* state is responsible for responding to clients and updating the new members to the current state of the Objectpool. The token is passed among the full members of the group every time a data packet is received. The frequency with which a token is passed can be adjusted, say after n data packets are received. A RMP handler is used as a *ROL Token*. After servic-

Event	Condition	Next State	Action(s)
Data Packet	(none)	Token Site	(i) Send Token Pass Packet to next site on token list. (ii) Reply to Packet if sender is client and needs response.
RMP_GRP_CHANGE	next token site requested the ROL Token	Not Token Site	(none)
RMP_GRP_CHANGE	new member added to the group	Token Site	Send the state of object group to new site.
RMP_GRP_REFORM	site still has the token	Token Site	(none)

TABLE 4 Token Site State

Event	Condition	Next State	Action(s)
RMP_GRP_REFORM	site doesn't have the ROL Token and is not at the head of member-ship list	Not Token Site	(none)
RMP_GRP_JOIN	(none)	Token Site	Send Update Pool Start Packet and transfer the state of objectpool to new member
Pass Token Alarm	(none)	Token Site	send Token Pass Packet again.

TABLE 4 Token Site State

ing a request from the client or another ROL server, the token site sends Token Pass packet to next ROL server in the token ring and releases the ROL Token. The next ROL server in the RMP token ring is determined using the RMP Token list. Upon receiving the Token Pass packet from the token site, the next ROL server requests for the ROL Token. The Pass Token Alarm and Recv Token Alarm timers are used in case a request for ROL Token fails. On expiration of the timers the request for ROL Token is made again. Note that there is a time difference between the time when token site relinquishes the ROL Token, and the time when next ROL server gets a token. The first site after relinquishing the ROL Token remains in the *Token Site* state and continues to service all the client requests till the next ROL server becomes the token site. RMP provides notification events, a RMP_GRP_CHANGE event to the first site and a RMP_GRP_RCV_LCK event to the would be token site. The first site transitions to the *Not Token Site* state and the would be token site transitions to the *Token Site* state. These two events are provided atomically by RMP, hence no client request goes unserviced or is serviced twice.

RMP notifies an application with a RMP_GRP_REFORM event when there is a site or a communication failure in the group and the RMP token ring undergoes successful reformation. An RMP reformation is successful if and only if the ring has the majority number of members in it and all the members have the same set of packets (and do not have any missing packets or holes in the OrderingQ). The presence of these packets means ROL has not lost any data packets during the RMP reformation. After RMP reformation, it might be possible that none of the sites has the ROL token. If the token site has the ROL token then it continues to be a token site. If the token does not have the ROL token then membership list is used to decide which site will be next token site. The member which is at the head of the membership list becomes the next token site. Thus, the virtual token ring algorithm is robust in face of RMP reformation. RMP reformation may take from some seconds to a few minutes depending upon the TTL value of the group. The ROL servers would not be able to respond to requests from clients during this time. Thus, clients may have to resend their requests.

A ROL server starts in the *Not in Group* state. A member transitions to the *Token Site* state if it forms an RMP Group, else it transitions to the *Joining Group* state when it receives the Update Pool Start Packet from the token site. Only one member can be in the *Joining Group* state at a time because the token site sends all state information to new member after sending an Update Pool Start Packet and only then it will service the next new member if any. The ROL token site transmits the state of the objectpool and the transactions to the member

Event	Condition	Next State	Action(s)
Token Pass	this site is the next token site	Not Token Site	try to acquire the token

TABLE 5 Not Token Site State

Event	Condition	Next State	Action(s)
RMP_GRP_RCVLCK	check whether this site got the token	Token Site	(none)
RMP_GRP_FORM	(none)	Not Token Site	request for the token
RMP_GRP_REFORM	no site in the group holds the token and this site is at head of membership list	Token site	request for the token
RMP_GRP_DENLCK	If this site has received token pass packet and is the next token site	Not Token Site	request for the token
Recv Token Alarm	(none)	Not Token Site	request for the token
Data Packet	(none)	Not Token Site	process the packet

TABLE 5 Not Token Site State

Event	Condition	Next State	Action(s)
Update Pool Start	(none)	Joining Group	(none)
RMP_GRP_FORM	(none)	Token Site	request for the token

TABLE 6 Not in Group State

in the *Joining Group* state using an unicast connection. The ROL token site does not pass the token until it transmits the complete state of object pool to the new member. No other requests are handled by the token site until the new member receives the complete state of the object pool. Thus, the new member gets the exact snapshot of the group's replicated objectpool. The member in the *Joining Group* (new member) processes the unicast data packets it receives from the token site and caches all the updates made to the object pool. The token site sends Update Pool End Packet after sending the complete state of the group to the new member. After receiving the Update Pool End Packet the new member processes

all the cached updates in order and transitions to the *Not Token Site* state. The above mentioned method of update is synchronous in nature, no other requests are handled by the token site until the new member receives the complete state of the object pool. The updates can also be done in a asynchronous manner, but it would result in more complex token ring and the recovery algorithms.

Event	Condition	Next State	Action(s)
Update Pool End	(none)	Not Token Site	(none)
Data Packet	(none)	Joining Group	cache the packet
Unicast Data Packet	(none)	Joining Group	process the packet

TABLE 7 Joining Group

Members in the ROL transition to the *Leaving Group* state before leaving the RMP Token ring. A member stays in this state until all the transactions it has participated in have been committed.i.e., the two requirements of minimum persistence and majority resilience are met for all transactions. RMP takes care of the second requirement in its model (if member is in leaving state) but the first requirement has to be taken care of by this layer. Hence a member has to stay in the RMP ring until the first requirement of minimum persistence is met.

Upon receiving the notification from the application, the ROL member sends a state packet stating that its leaving the object group and transitions to the *Leaving Group* state. Other members of the object group update their state lists. All the commits taking place after the

timestamp of the state packet do not take this member into account for persistence. Once

Event	Condition	Next State	Action(s)
Leaving Group Alarm	Minimum persistence not met for all transactions	Leaving Group	(none)
Leaving Group Alarm	Minimum persistence met for all transactions	Not in Group	Leave RMP ring

TABLE 8 Leaving Group

the minimum persistence is met for all the transaction commits it has participated in, the member leaves the RMP ring.

3.4.2 Reformation Extension of ROL

RMP reformation algorithm fails if majority or more than majority of members in the ring have failed, or if it cannot recover all the packets (the OrderingQ has some holes) during reformation. The first case is treated as a failed reformation and the application is notified with the RMP_FAILREF event. The second case is treated as a atomicity violation and application is notified with the RMP_ATOMV event. RMP stops all its processing after failed reformation or atomicity violation. ROL provides reformation extensions to provide recovery mechanism from RMP reformation failures.

Three more states are added to support the Reformation extension in ROL. The three new states are the *Start Recovery*, *Reform* and *Synch Commits* states. After a failed reformation the ROL Server transitions to the *Start Recovery* state from all normal operation states except the *Not in Group* state. The position of the member in the token list is stored in the membership list. The updates are made to the membership list only in normal operation

states. The reform site is the surviving member which is at the head of the list and it forces the formation of the ring. All the other members try to join the ring after a predetermined

Event	Condition	Next State	Action(s)
RMP_FAILREF	Reform site	Start Recovery	Reform site forces formation of RMP ring
RMP_ATOMV	Reform site	Start Recovery	Reform site forces formation of RMP ring
RMP_FAILREF	Not a reform site	Start Recovery	After a predetermined time-out try to join RMP ring
RMP_ATOMV	Not a reform site	Start Recovery	After a predetermined time-out try to join RMP ring

TABLE 9 All Normal Operation States

Event	Condition	Next State	Action(s)
RMP_GRP_FORM	(none)	Reform	(none)
RMP_JOIN	(none)	Reform	(none)

Table 10: Start Recovery State

Event	Condition	Next State	Action(s)
Reform Alarm	Reform site	Reform	Ping for other groups, if there exists one with greater or equal no. of member, send abort request
Reformation Abort	(none)	Start Recovery	(none)
Synch Commits	(none)	Synch Commits	(none)

TABLE 11 Reform State

time-out to avoid race conditions. The time-out is dependent upon the position of the mem-

ber in the token list. The formula tries to avoid formation of multiple rings by making the time-out a multiple of time taken to form a ring which is dependent on TTL value of the

time-out = position of the member in the token list saved before failed reformation * timetaken to form a ring (TTL value of the group)

group. But, the formula doesn't entirely remove the race conditions. A random factor should be introduced to avoid the race conditions.

The ROL member transitions to the *Reform* state from the *Start Recovery* state when it joins or forms the ring. The reform site in the *Reform* State is the member which is at the head of the RMP token list.The members in the *Reform* state transition into the *Synch Commits* state when the two conditions are met. The first condition is that the majority number of the ROL members from the old ring should be present, so that only one partition functions after reformation. The second condition is that one of the members in the reformed ring has all the commit messages that have occurred before the failure took place. If we keep the minimum persistence in a group fixed say P where the value of P is between 1 and RMP Majority -1. Then the last transaction to be committed would have been flushed to at least P sites. And these P sites have all the commit messages including the updates made by the transaction flushed to disk before the last commit message, since we are making the assumption that all the flushes of commits will be in order of their timestamps. So, for a group with N members, one of the N - P + 1 sites must have all the commit messages.Hence if minimum size is set to N - P + 1, then the last commit message before the failure is MAX (PersistenceTSP at all sites). Lets define the term *Commit Majority* as N - P + 1, which is number of members required to satisfy the two conditions given above.

In the Reform state there are two possibilities:

(i) Ring has Commit Majority: The membership list contains the state of the group before reformation. The reform site in the ring will wait till (no. of the members in the old token list * time taken to form the ring) time so that all the members have exercised their time-outs to join the ring. The reform site will then send Synch Commits Request packet to the group and all members transition to the *Synch Commits* state.

(ii) Ring doesn't have a Commit Majority: The reform site pings the other rings with the same multicast address, port and groupname. If the number of ROL members (which were in the old ring) present in the other group is greater or equal to the number of members in the group then it will send Reformation Abort Request to the group. All the members in the group transition to *Start Recovery* state.

Event	Condition	Next State	Action(s)
Unicast NACK packet	(none)	Synch Commits	Send the commit
Resume Normal Operation	Reform Site	Token Site	(none)
Resume Normal Operation	Not Reform Site	Not Token Site	(none)

TABLE 12 Synch Commits State

In the *Synch Commits* state the members use NACK packets to recover the missing commits. Once all the members have synched to the commit synch point the reform site sends Resume Normal Operation packet to all the other group members to resume normal operation.

If all the sites have failed then a special start-up procedure is needed. The start-up procedure consists of reading the membership lists and the objectpool information from the stable store to the memory and then transitioning to the *Start Recovery* state. Then the sites use

the recovery algorithm described in the section to recover all the commits and return to the normal operation mode.

Chapter 4 Implementation of Replicated Object Layer

This chapter discusses the implementation details of the Replicated Object Layer. ROL provides an object-oriented framework to replicate data on multiple sites and across different platforms. It provides a type registration layer that allows the application to register platform independent types. ROL also provides support for performing distributed atomic transactions on replicated data. ROL being a middleware, provides an application programming interface for applications to be written on top of it. ROL hides the burdensome details of replica consistency operations and makes it transparent to the application.

ROL was implemented using C++ [Str86] language. C++ has almost all the important features used in object oriented programming, and efficient implementations of the C++ compiler are available on all the major platforms. The event driven paradigm was chosen for implementing the distributed applications on top of ROL. The application constructs a control loop and passes control to ROL for its internal processing. This scheme allows the applications to have explicit control over when ROL gets to perform its operation. ROL notifies the application in form of events.

4.1 ROL Class Structure

The ROL class structure and modules for the implementation can be divided into three basic categories:

- Type Registration
- Distributed Transactions
- Application Programming Interface

4.1.1 Type Registration

ROL provides a type registration layer for application to create platform independent types. Sun's External Data Representation (XDR) [Sun87] is used at the presentation layer to ship the data across different platforms. XDR was chosen as it is one of the defacto standards and one has to only do one step conversion that is convert the data from local machine format to XDR format and vice versa.

4.1.1.1Type Class

This class is used to register typing information for a replicated object. The ROL supports the basic types like integer, character, float etc. It also supports the composite types arrays and structures. The basic types are created and initialized by the ROL system. Once the initialization is done the user is not able to create basic types and can only create composite types. Each type requires an XDR function which encodes and decodes the data from the local machine format to XDR format. As of now, the user is required to pass the XDR function as a parameter while creating a type. Sun's RPCGEN compiler is used to simplify the creation of XDR functions. In future the generation of this function can automatized.

Composite Types

ROL system supports two composite types, arrays and structures. Composite Types of arbitrary depth can be created on ROL system, which means that it is possible to create an array of structures where the structure may contain array members and so on.

Arrays: ROL system supports creating array types. An array type can be created by specifying the reference or child type of the array and the number of elements in the array.

Structures: A list of members and their respective offsets in the structure are needed to create a structure type.

4.1.1.2Field Class

Field class is used to describe a field of data in a given type. It specifies the type of the chunk of data and its offset in the object. This class is used in creating structures and, reading and writing data from the replicated objects.

4.1.2 Distributed Transactions

ROL provides applications the capability to perform distributed atomic transactions on replicated data. An efficient locking scheme is provided to preserve the consistency of replicated data in this layer. The lock and update messages on replicated objects are sent to the group as totally ordered messages.

4.1.2.1Lock Class

This class provides multigranular read/write locks on the replicated data. A transaction can lock the data items depending upon the granularity it operates on. A transaction can lock part of the data object instead of the complete object. This allows two transactions access-

ing the different fields of the same object to run concurrently (or in an interleaved fashion to be more precise) without blocking one another. For example, two transactions writing different elements of an array (and assuming their read/write locks don't conflict) can run concurrently by locking just the array elements (instead of whole array). The Multigranularity Locking (MGL) protocol [BHG 87] is implemented to maintain the consistency of the data. The lock object defines a lock table for a data type. The implementation of the lock table is done using trees.

4.1.2.2Update Class

This class contains the information of which part of an object is to be written or locked/ unlocked. If the update is a write, then it contains information whether the write has been applied to the object in memory and stable store.

4.1.2.3CommonLog and Log Classes

The Common Log and Log Classes are the container classes of update objects. All the updates made by the transactions are applied to the objects only when the transaction is committed. All the information about the updates is kept in form of Log and CommonLog objects. Log object is the log of updates associated with a particular transaction. It is implemented as a linked list of updates. The CommonLog is the log of updates going on in an objectpool. CommonLog is essentially a hash table of list of updates, hashed by the object ID. CommonLog contains the global order of updates made by different transactions in an objectpool. Log object is provided to implement fast commits as the time taken to access an update in CommonLog object is much longer.

4.1.2.4Transaction Class

A transaction can be done on ROL system by using the transaction object. A transactionID is associated with each transaction. All the writes and locks to an object can be associated

to a transactionID. The consistency of data can be maintained across multiple sites using transactions. A transaction can be aborted or committed at the end. The transaction class contains the Log object and all the updates made by the transaction are logged on to the Log Object. When the transaction is committed the updates are written to the object else the updates are thrown away. Transaction time-outs can be specified so that the transaction is timed out after sometime and locks are released. This reduces deadlocks as mentioned in Section 2.6.1.

4.1.3 Application Programming Interface

4.1.3.1Object Class

The Object class corresponds to the replicated object in ROL. The replicated object contains the type of data, the data itself and lock information on that data. The lock information for the object is held in the lock table. The object interface allows the user to read from or write to or lock/unlock data fields of a replicated object. The interface is very transparent to the user. Each replicated object has a UniqueID called ObjectID associated with it. Since objects are persistent they are unique with respect to space and time. The UniqueID is made up of Process ID which uniquely identifies the RMP process in the Internet, current time in microseconds and a counter to differentiate the objects created in the same microsecond. The size of the UniqueID is 20 bytes.

4.1.3.2Objectpool Class

The replicated objects are grouped into pools called Objectpools. Each Objectpool is associated with a RMP Group or RMP Token ring. Objects in this pool are replicated to each ROL Server of this group. It is a container class of objects, transactions and types. This class also acts as a communicator class. It also contains the commonLog object which con-

tains all the updates made to the object. It provides an interface to create objects, define types, begin and commit transactions, and register callback functions in the objectpool.

4.1.3.3Client Class

The Clients use Multi-RPC to get information about an object in a pool. The clients don't store any object information with them and use object ID's to access objects in the replicated pools. The client class provides same interface as Objectpool and Object classes for the client side. The ROL Token ring takes some seconds to few minutes for reformation (because of site and communication failures) and the ROL Servers don't respond to client requests during this time. It's possible that ROL Servers don't service a client's request. The clients have to resend requests after some time-out period in such cases. But resending the requests has also another problem, a server may reply to client's request twice. The client is made robust to handle these failures.

4.1.3.4ROLEvent Class

The ROL layer uses ROLEvent to notify the application about the changes made by other sites in the replicated objectpools. These events are generated when:

- object or type is created or deleted.
- an update is made to an object.
- a lock is released or acquired.
- a transaction is started or committed.

Some of the calls in the ROL API are asynchronous in nature. The events are used to notify the application upon success/failure of an operation using ROLEvents. The ROLEvents

also give additional information to the application. For example, when an object is created, the pointer to object is returned to application.

4.2 An Example: A Simple Replicated Database Application

This section describes a simple replicated database application implemented on ROL. This application stores employee records. The application maintains a linked list of employee records.

The application creates an ObjectPool object as the first step. The RMP groupname or IP multicast address of the group, TTL value of the group, fault tolerance of the ring and a flag (specifying whether to form a group or join an already existing one) are given as parameters to the constructor of ObjectPool object. The fault tolerance of the ring is minimum number of members that should be present in the ring for the reformation to be successful. Depending upon the flag given to the constructor an RMP ring is formed or an already existing ring is joined. The application should give control to the ROL regularly to do its internal processing. This is done by using an event loop style of programming similar to X-Windows [SWG86]. In the event loop the application gives control to ROL and can do part of its processing. The ROL does its internal processing and notifies application using ROL Events. The application can set callback functions to be invoked upon notification of a ROL Event. Our application sets appropriate callback functions to be invoked by ROL.

Now that application has joined a ring, its time to define types to be used by it. Let's say employee record is a structure type with employee's name (a 40 character array), employee's age (an integer type) and salary (a float type). The applications access the types and objects using their UniqueIDs. One way of sharing types and objects by the application at different sites is to negotiate or decide the values of UniqueID's at the compile time. An-

other way of sharing objects which is at run time is by having application keep a map from UniqueID to a field in the object. Both the methods are used by this example. Two types (name array and the employee structure) are defined using two UniqueIDs known at compile time. These two types are defined by the member that forms the group. The other members that join the group need not define these types.

Now that the types have been defined, its time to create replicated objects in the group. An object can be created by invoking the createObject function in the ObjectPool object and passing the ID of the type which is employee structure here. The ROL system returns the ID of the object. The application can now write to the replicated object using write operations, so the name, age and salary fields are written appropriate values. The application keeps the mapping from ID of the object to name of the employee.

Lets assume that application running on site A formed a group and created an employee object for employee named Sam. Then, the application running on site B joins the group. It receives all the types, objects and updates made to the objects in the group and all this is transparent to the application. The application is notified about the new types, objects and updates in form ROLEvents. When the site B receives the object created at site A, the ROL notifies the application at site B with ROL_CREATE_OBJECT event. The callback function which was registered for this event is invoked. In this function the application gets the other information from the event which is the object itself. The application at site B reads the name of the employee from the object using read operation and keeps a map between the object and the name. Now, both the sites A and B have and know about the employee object with the name Sam. Any changes made by the application one site are notified to the applications at other sites. If site B makes an update to age field for the Sam employee ob-

ject, it is sent as totally ordered messages to the group and the application at other sites are notified with ROLEvent ROL_WRITE.

For more complex updates, distributed transactions provided by ROL can be used. Let's assume that application on site A and site B attempt to make updates to the Sam object simultaneously. Both site A and site B can begin a transaction and try to get locks on Sam object. If site A succeeds in getting the lock for the Sam object then site A is notified about the success and site B is notified about failure using ROL_LOCK event. Site A has exclusive access to the Sam object, makes writes to it and commits the transaction. Site B can either abort the transaction or wait for some time before requesting for lock on Sam object again.

```
create objectpool // form or join a RMP token ring and get the state
                  // of the objectpool if a ring is joined
register callback functions for ROLEvents
// event loop
while (not done) do
    give control to ROL //  for ROL internal processing. ROL
                  // makes callbacks on receipt of ROLEvents
    do application processing // can make RMP API calls
od
call objectpool destructor
```

Figure 6 ROL Server procedure

The figure 6 shows a sample server procedure. An objectpool is created and the callback

functions for certain ROLEvents are registered. The application then enters an event loop. The control has to be given to ROL in the event loop for its internal processing. ROL makes appropriate callbacks upon the reception of ROLEvents. The application can do its processing in the loop, but should return the control to ROL as soon as possible. The figure 7

```
create a client object // establish connection to a RMP token ring
define types, create objects in the object server group
begin a transaction
make updates to the objects // objects are accessed using their ID's
commit the transaction
....
```

Figure 7 ROL Client Procedure

describes the sample ROL client procedure. A client object is created to establish connection to a RMP token ring. A client can then define types and create objects. The server group returns ID's of the types and objects created to the clients. The clients access the objects in the server group using their ID's. A client can do updates on the objects and perform distributed transactions.

The application doesn't have to do any work in case of site failures, the reformation is handled transparently by the ROL. Let's assume that application is running on three sites A, B and C and fault tolerance of the group is set to majority which is 2 in this case. If site B fails, the ring undergoes reformation and ROL recovers transparently to the application.

Chapter 5 Verification of ROL

This chapter describes the approach taken to verify the ROL using the SPIN tool. A SPIN model is developed for ROL to check the consistency and correctness of its state specifications.

5.1 SPIN Tool

SPIN [Holtz94] is a tool for analyzing the logical consistency and general verification for proving correctness properties of distributed and concurrent systems, especially for data communication protocols. The system is described in a modeling language called PROMELA. SPIN provides support for processes which exchange messages through communication channels. Communication via message channels can be defined to be synchronous(i.e. rendez-vous), or asynchronous. The language allows for dynamic creation of processes. The protocol system is described as a group of processes running at their own rate, exchanging message through communication channels. Each process can make transition based on state variable values and channel event and produce output to other processes' communication channels.

Given a model system in PROMELA, SPIN can either perform random simulations of the sesames execution or it can generate a C program that performs a fast exhaustive validation of the system's state space. During simulations and validations, SPIN checks for the absence of deadlocks, unspecified receptions and inexecutable code. The validator can also be used to verify the correctness of the system invariants specified as never clauses, and it can find non progress execution cycles and livelocks. SPIN has adopted some advanced algorithms to address the state explosion problem. Users can use either state reduction algorithm or bit-state reduction to perform the best possible search in the case of state explosion.

5.2 SPIN Model for ROL

The SPIN tool is used to construct a multiple site interaction model. The model decibels interaction between ROL processes at multiple sites and a RMP process providing the reliable multicast communication to all ROL processes. The ROL processes communicate with each other using ROL Control Packets. The ROL processes communicate with RMP process using the RMP API calls and RMP process responds back by sending appropriate RMP events to ROL processes. The ROLEvent queue and RMP API call queue are modeled as communication channels. The states of ROL processes, the number of sites in the token ring and the current tokensite are stored as state variables.

The model works as follows. Upon the reception of an ROLEvent and a condition being true a ROL process may transition to another state and may invoke an action(s). The actions are the RMP API calls posted to RMP process's communication channel. The RMP process polls the RMP API calls from the RMP API call queue. The RMP process does the appropriate processing (i.e, make changes to the values of state variables in the process) required

by the API call and then posts appropriate RMPEvents to the ROL processes. The ROL processes can also post unicast ROL Control packets to each other.

An example is given to demonstrate the functioning of SPIN model for ROL. Consider a model with two ROL processes A and B. Let's assume process A is in the *Token Site* state. Upon reception of ROL Data packet from process B, process A attempts to pass the token to the process B by posting Token pass packet in it's ROLEvent queue and releasing the ROL token using the appropriate RMP API call. The process B upon the reception of Token

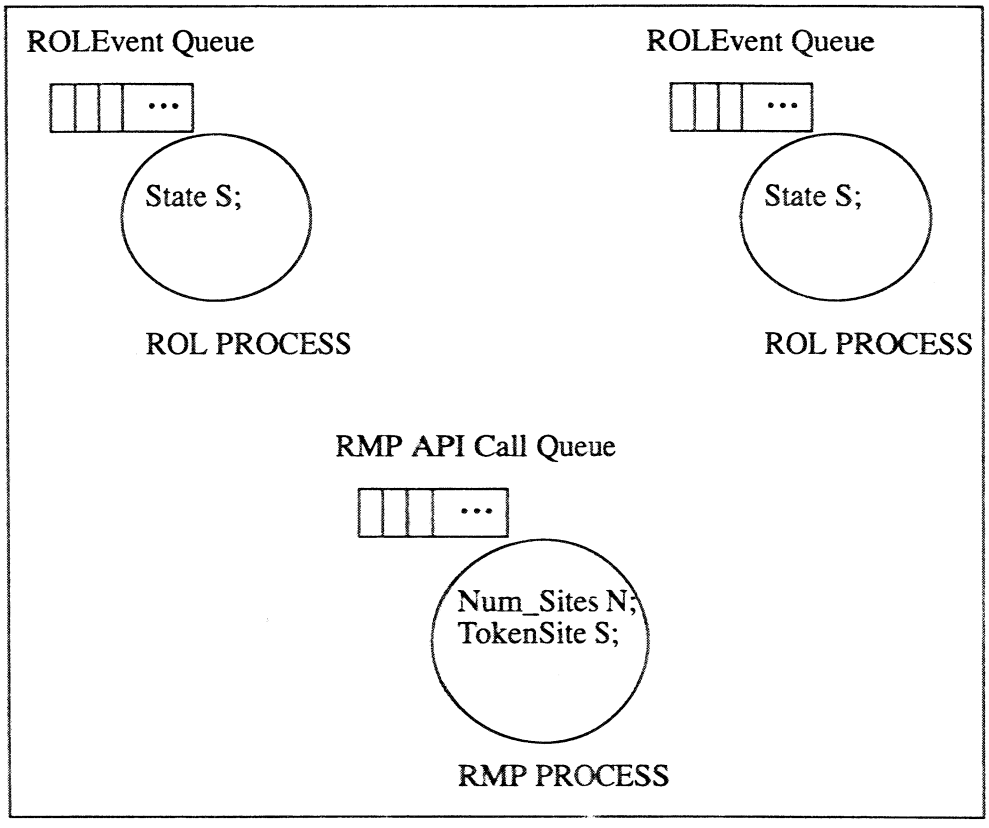

Figure 8 SPIN model for ROL

pass packet posts appropriate RMP API call to get the ROL token. RMP process polls the API calls from the queue, processes them and posts appropriate ROLEvents in the ROLEv-

ent queue. It posts RMP_GRP_CHANGE and RMP_GRP_RCVLCK events respectively in both the processes' event queues. The processes A and B switch to the Not Token Site and Token Site states respectively upon the reception of RMP_GRP_CHANCE and RMP_RCVLCK events. RMP process must generate the RMPEvents corresponding to RMP API calls. The correctness of the SPIN model depends upon the correctness of the mapping from RMP API calls to RMP Events.

Thus, the multiple site interaction model can be simulated using the SPIN tool and checks can be made for system invariants, deadlocks and livelocks.

Chapter 6 Conclusions and Future Work

This chapter draws some conclusions about the ROL's design and implementation. It presents an outline for possible future research.

6.1 Conclusions

Presented in this document are the design and implementation aspects of the Replicated Object Layer. To summarize:

- An object oriented framework for replicating data on multiple sites and across different platforms is presented.
- The system frees application from burdens of performing replication consistency and fault tolerance protocols for maintaining the consistency of data. The system does all the work which is completely transparent to the application.
- A commit protocol to commit transactions efficiently in the RMP environment is presented.
- The algorithms for membership changes in the group, virtual token ring and recovery are specified using finite state machine approach.

- Reliable Multicast Protocol provides excellent support for building fault tolerant applications.

6.2 Future Work

The directions for further work are:

- ROL provides strong consistency model for replicating data using totally ordered updates and distributed atomic transactions. A weaker consistency model could be developed giving unique id's to updates and using a negative acknowledgment scheme for missing updates.

- Some optimizations and improvements can be made to algorithms presented in ROL.

  - Virtual Token Ring algorithm can be improved by using multiple handlers provided by RMP, instead of the one handler being used currently.

  - Recovery algorithm can be improved by introducing randomness in the time-outs used in *Start Recovery* state.

  - Update of the objectpool state to a new member can be made asynchronous.

- A transaction flow control mechanism can be developed wherein the amount of updates taking place in the ring and the speed of stable store processing can control the rate of transactions.

Bibliography

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987

[Bir93] The Process Group Approach to Reliable Distributed Computing. *Communications of ACM*, 36(12):37-53, December 1993.

[CM84] J.M. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251-273, August 1984.

[Deer89] S. Deering. Host Extension for IP Multicasting. Technical Report RFC-1112, IETF, June 1988.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger. The Notion of Consistency and Predicate Locks in a Database System. Communications of ACM, Vol. 19, No. 11, November 1976.

[Gray78] J.N. Gray. Notes on Database Operating Systems. *In Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, volume 60, pages 339-481. Springer-Verlag, Berlin, 1978.

[Hans94] Erikson Hans. "MBone: The Multicast Backbone". Communications of the ACM, August 1994, Vol.37, pp.54-60.

[Holtz94] G.J. Holtzman. Basic Spin Manual. AT&T Bell Laboratories, Murray Hill, New Jersey, 1994.

[Lit91] Mark C. Little. Object Replication in a Distributed System. Ph.D. Thesis, University of Newcastle Upon Tyne. September 1991.

[MCW95] T. Montogomery, B. Whetten & J. Callahan. "The Reliable Multicast Protocol Specification", West Virginia University, Morgantown, October 5, 1995.

[Mont94] T.Montgomery. Design and Implementation, and Verification of Reliable Multicast Protocol. MS thesis, Department of Electrical and Computer Engineering, West Virgina University, Morgantown 1994.

[Moy94] J. Moy. Multicast Extensions to OSPF. Technical Report RFC-1584, IETF, Mar 1994.

[Ske82] D. Skeen. A Quorum-Based Commit Protocol. *In Berkeley Workshop on Distributed Data Management and Computer Networks*, number 6, pages 69-80, February 1982.

[SS83] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9 No. 3,May 1983.

[Sun87] Sun Microsystems 1987, "XDR: External Data Representation Standard". RFC 1014, 1987.

[SWG86] Scheifler, Robert W. and James Gettys. "The X-Window System". ACM Transactions on Graphics, No. 5, pages 79-109, April 1986.

[Str86] B. Stroustrup, "The C++ Programming Language", Addison Wesley, 1986.

[WPD88] D. Waitzman, C. Partridge and S. Deering. Distance Vector Multicast Routing Protocol. Technical Report RFC-1075, IETF, November 1988.

APPENDIX The Replicated Object Layer Application Programming Interface

The Replicated Object Layer(ROL) is an object oriented service for replicating objects in a distributed system. ROL provides basic object type registration and the capability to create and update the replicated objects. ROL provides basic transaction capability through multigranular read/write locks on the individual fields of an object and maintaining a log of updates. ROL uses the Reliable Multicast Protocol(RMP) as communication protocol that provides for reliable delivery, serialization and casual ordering. In ROL, objects are grouped into object pools that are replicated across a set of processes at different sites in a network.

This layer provides user a basic typing facility to create types. These types are used to create replicated objects using Object and ObjectPool classes. Each replicated object is associated with an UniqueID. Since these objects are persistent the UniqueID is made unique with respect to space and time. ROL uses the XDR functions to ship the data across different platforms. The fields of an object can be updated by read and write operations. The write operation makes updates on all copies of the object whereas the read operation reads the local copy. The transactions can be implemented on replicated objects using the BeginTransaction and EndTransaction operations. The locks on individual fields on an object can be acquired using the lock operation. The transaction can be committed or aborted using EndTransaction operation. A log of updates is kept for each Object in an ObjectPool which are used to implement the transactions.

ROL has two kinds of members, servers and clients. The servers of a group maintain the object pools for that group. The clients of a group do not belong to that replicated object group and do not maintain the object pools for that group. They use one of the servers to do transactions on an object. A virtual token ring is implemented in the ROL layer to handle the client requests and new servers. A token is passed around the ROL servers and the tokensite handles the client requests and brings a new member to state of the group transparent to the application. ROL is an event based system that notifies the application using events and provides asynchronous non-blocking calls.

TYPES AND CLASSES

Various types and classes in ROL API:

Classes:

Type - Type object containing information of a type.

Field - Field object containing information about a field in an object.

Object - replicated object

ObjectPool - Container for replicated objects. ROL control object for servers

Client - ROL control object for clients.

ROLEvent - notifies application of asynchronous events.

Pointers:

TypePtr - pointer to Type object.

FieldPtr - pointer to Field object.

Structures:

xdrFuncptr - bool_t (*)(XDR *, Value, TypePtr);

UniqueID - Unique ID for replicated objects. Also used for types and transactions.

```
struct UniqueID {
        struct RMPProcID procID;
        struct timeval time_val;
        short count;
};
```

Value -

```
typedef union
  {
```

```
        void *ptr;

        void **dptr;

}Value;
```

OBJECT CLASS

The Object class represents the replicated object. It contains the data of the replicated object, the type information and locks for various fields of the object.

Constructor and Destructor

Access is not given to the constructor and destructor of Object Class as ObjectPool class manages Object class objects. ObjectPool Class provides createObject and deleteObject functions to create and delete objects respectively.

Reading a field of an Object

```
int Object::read(Field field, void *value)

int Object::read(Field field, void **value)
```

Description: This function is used to read a member from a local object. The second version is used to read a member whose XDR function needs a double pointer to be passed to it.

Arguments: The first argument is the field of the member to be read and the second argument is the address of the variable the value of member is read into.

Returns:

Code	Value	Description
ROL_OK	1	success.
ROLEXDRDE	-102	XDR Decoding Error

NOTE: The user should make sure that memory is allocated for the second argument before calling the read function.

Writing a field of an object

```
int Object::write(Field field, void *value, UniqueID *tid=NULL)

int Object::write(Field field, void **value, UniqueID *tid=NULL)
```

Description: This function is used to write a member on all copies of an object when the write is not associated with any transaction(tid is NULL). When associated with a transaction the write will add the update to the log which will be committed or aborted at the end of transaction. The second version is used to write a member of type array to a local object.

Arguments: The first argument is the field of member to be written and the second argument is the address of the variable whose value will be assigned to the member. The tid is the transaction ID of the transaction this write is associated with.

Returns:

Code	Value	Description
ROL_OK	1	success
ROL_EXDREN	-101	XDR Encoding Error
ROL_EXDRDE	-102	XDR Decoding Error
ROL_ESMSG	-103	Error sending the message to other ROL Servers.
ROL_EOFF	-202	Negative offset specified in the field.

Acquiring or releasing a lock on a field of an object

bool Object::fieldlock(FieldPtr field, UniqueID *tid)

Description: This function is used to lock a member of an object. The field is not locked if it and its sub members are not locked by any other transaction. This is an asynchronous non blocking call. The result of the lock operation is notified by the ROL_LOCK event.

Arguments: The first argument is the field of a member to be locked and the second argument is the ID of the transaction.

Returns:

Code	Value	Description
ROL_OK	1	success
ROL_EXDREN	-101	XDR Encoding Error
ROL_EXDRDE	-102	XDR Decoding Error
ROL_ESMSG	-103	Error sending the message to other ROL Servers.
ROL_EOFF	-202	Negative offset specified in the field.

Object Access Methods

UniqueID *Object::getID()

Description: returns the Object ID.

Arguments: none

Returns: ID of the object.

OBJECTPOOL CLASS

The ObjectPool object is the container of replicated objects. Each ObjectPool is associated with a RMPGroup. It uses the RMPGroup object for group communication whose name is the argument to the constructor. An ObjectPool is a pool of objects. It is the unit of replication in Replicated Object Layer. It also maintains the details of transactions.

Constructor and Destructor

ObjectPool::ObjectPool(RMPGroupID *grpid, RMPProcID *proxy, n_char ttl,

n_char minsz, n_long flags, struct RMPopts *optvals)

Description: Creates an ObjectPool object.

Arguments: Same as for MLRGroup::Join method. When proxy is NULL it reduces to shorter version of MLRGroup::Join method. If MLRControlLoop object is not initialized using ObjectPool::init method, it is done now before creating a new group.

Error Codes: Static member ObjectPool::_serrno will be set to proper error code.

Code	Value	Description
ROL_OK	1	success
ROL_EARG	-105	grpid is NULL
ROL_EXDRDE	-403	ObjectPool with that groupid already in existence
ROL_ESMSG	-405	Error initializing the RMP object
other codes		returned by MLRGroup::Join method

~ObjectPool::ObjectPool()

Description: Destroys the ObjectPool object.

Arguments: None.

Error Codes: Static member ObjectPool::_serrno will be set to proper error code.

Code	Value	Description
ROL_OK	1	success
other codes		returned by MLRGroup::leave method

Initializing a RMP Process

static int ObjectPool::init(struct RMPProcID *id, n_long flags,

n_long mask = 0, char *file = "stderr")

Description: This function initializes the MLRControlLoop Object. It must be called only once.

Arguments: Same as MLRGroup::init method

Returns:

Code	Value	Description
ROL_OK	1	success
ROLEALLOC	-404	grpid is NULL
ROLEDEL	-401	ObjectPool with that groupid already in existence
ROL_EBUF	-402	Error initializing the RMP object
ROL_EINDN	-413	initialization already done
other codes		returned by MLRGroup::init method

The following are the possible error codes that can be set by the remaining methods in this class. Many of the error codes are common to most of the methods. Hence they are not listed individually for each function.

Possible Error Codes: Member ObjectPool::_errno will be set to proper error code.

Code	Value	Description
ROL_EXDREN	-101	XDR Encoding Error

Code	Value	Description
ROL_EXDRDE	-102	XDR Decoding Error
ROL_ESMSG	-103	Error sending the message
ROL_EARG	-105	Invalid arguments passed to the method
ROL_EOBJEX	-406	Object already exists
ROL_EOBJNEX	-407	Object doesn't exist
ROL_ETNEX	-408	Transaction doesn't exist
ROL_ETYNEX	-409	Type doesn't exist
ROL_ETYEX	-410	Type exists
ROL_EREFNEX	-411	Reference Type doesn't exist
ROL_EFOFF	-412	Field offset is negative

Defining a type

```
TypePtr ObjectPool::defineType(kind op, int size, char *tag, FieldPtr flist,
        TypePtr ref, xdrFuncptr xdrfunc, UniqueID *type_id = NULL,
        xdrproc_t xdrchildproc = NULL)
```

Description: This function is used to create a type.

Arguments: op is the kind of type(described in Type::findtype function). tag is used for structures to distinguish two with same size. flist is the list of members in a structure. ref is the child type for arrays and xdrchildproc is the XDR function for that type. xdrfunc is the XDR function for the type. A type_id can also be passed if type_id is known beforehand.

Returns: Pointer to type on success else NULL.

Error Codes: See the possible error codes above and the error codes returned by Type::init method if the return is NULL.

Getting reference to an already existing type

```
TypePtr ObjectPool:: getType(UniqueID *type_id, int op)
```

Description: This function is used to get a type from the ObjectPool.

Arguments: ID of the type and kind(refer to Type::init method) of type.

Returns: Pointer to type on success else NULL.

Error Codes: See the possible error codes above if return is NULL.

Deleting a Type

int ObjectPool::deleteType(UniqueID *type_id, int op)

Description: This function is used to delete a type from the ObjectPool.

Arguments: ID of the type and kind(refer to Type::init method) of type.

Returns: ROL_OK on success else ROL_ERROR.

Error Codes: See the possible error codes above if return is ROL_ERROR.

Creating an object in a ObjectPool

Object *ObjectPool::createObject(TypePtr type, UniqueID *object_id = NULL)

Description: This function is used to create an Object in the ObjectPool.

Arguments: The type of the object and the object ID(optional, if not specified ROL generates a UniqueID).

Returns: The pointer to Object created if successful. NULL in case of failure.

Error Codes: See the possible error codes above if return is NULL.

Getting reference to an object given its ID

Object *ObjectPool::getObject(UniqueID *object_id)

Description: This function is used to get an Object in the ObjectPool given its ID.

Arguments: the object ID

Returns: The pointer to Object created if successful. NULL in case of failure.

Error Codes: none

Deleting an Object from a ObjectPool

int ObjectPool::deleteObject(UniqueID *object_id)

Description: This function is used to delete an object from the ObjectPool.

Arguments: ID of the Object.

Returns: ROL_OK on success else ROL_ERROR.

Error Codes: See the possible error codes above if return is ROL_ERROR.

To start a transaction

UniqueID *ObjectPool::beginTransaction()

Description: This functions is used to start a transaction.

Arguments: none

Returns: ID for the transaction on success else NULL.

Error Codes: See the possible error codes above if return is ROL_ERROR.

To end a transaction

int ObjectPool::endTransaction(UniqueID *tid, bool commit_flag)

Description: This function is used to commit or abort a transaction.

Arguments: The transaction ID and commit_flag whose value should be TRUE to commit and FALSE to abort.

Returns: ROL_OK on success else ROL_ERROR.

Error Codes: See the possible error codes above if return is ROL_ERROR.

Granting Control to RMP

ROLEvent *ObjectPool::poll()

int ObjectPool::select(int width, fd_set *inreadfds, fd_set *inwritefds,

fd_set *inexceptfds, struct timeval *timeout)

Description: These functions are used to give control to RMP. The control must be returned to RMP regularly by calling these functions. All the ROLEvents to the objectpool are stored in FIFO queue. The poll method returns the ROLEvent in front of this queue.

Arguments: Same as for MLRControlLoop::select and MLRControlLoop methods

Returns: Same as for MLRControlLoop::select method. The poll method returns ROLEvent object.

Registering callback functions with ROL

int ObjectPool::setEventHandlers(ROLEventType, void (*func)(ROLEvent *))

Description: This function is used to set callback functions for a given event type.

Arguments: The first argument ROLEventType defines type of the ROLEvent.

ROLEventType	Description
ROL_CREATE_TYPE	a type is created in the objectpool
ROL_CREATE_OBJECT	an object is created in the objectpool
ROL_WRITE	an update is made on an object in the objectpool
ROL_DELETE_OBJECT	an object is deleted from the objectpool
ROL_BEGIN_TRANSACTION	a transaction is began in the objectpool
ROL_END_TRANSACTION	a transaction is ended in the objectpool
ROL_TRANS_WRITE	an update associated with a particular transaction is made in the objectpool
ROL_LOCK	a request is made to acquire or release locks in the objectpool
CALL_ALL_ROL_EVENTS	all the events above

The second argument is the pointer to event handler.

Returns: ROL_OK on success else ROL_ERROR for invalid event type.

Error Codes: none

Reclaiming a ROLEvent

int ObjectPool::reclaimROLEvent(ROLEvent *event)

Description: This function reclaims the ROLEvent object. This function must be called on each ROLEvent returned to application by ROL, else the same event is returned backed to application causing error.

Arguments: ROLEvent object

Returns: ROL_OK on success else ROL_ERROR

Error Codes: none

Getting access to control object in Messaging Layer

MLRControlLoop *ObjectPool::returnControl()

Description: This function returns the MLRControlLoop object if the application needs to access the MLRControlLoop interface.

Arguments: none

Returns: MLRControlLoop object

Error Codes: none

Accessing error codes

int ObjectPool::perror()

Description: This function returns the value of _errno member.

Arguments: none

Returns: value of _errno

CLIENT CLASS

The methods of client object are very similar to that of ObjectPool object. In addition the read, write and lock operations are provided by the client since clients don't have any persistent objects.

Constructor and Destructor

Client(MLRControlLoop *control, struct RMPGroupID *id, struct RMPProcID *proxy, n_char ttl, n_long flags, struct RMPopts *optvals)

Description: Creates an ObjectPool object.

Arguments: Same as for MLRGroup::MRPCconnect method. When proxy is NULL it reduces to shorter version of MLRGroup::MRPCconnect method.

Error codes: none

The following are the possible error codes that can be set by the remaining methods in this class. Many of the error codes are common to most of the methods. Hence they are not listed individually for each function. The errors could have occurred on client or server sides.

Possible Error Codes: Member Client::_errno will be set to proper error code.

Error codes for errors on the Client:

Code	Value	Description
ROL_ERMSG	-501	Error receiving message from the server
ROL_EMFAIL	-502	Failed to receive reply from the server
ROL_ESMSGS	-503	Error sending message to the server
ROL_EXDRENC	-504	XDR Encoding error on client side
ROL_EXDRDEC	-505	XDR Decoding error on client side

Error codes for Errors on the Server:

Code	Value	Description
ROL_EXDREN	-101	XDR Encoding Error
ROL_EXDRDE	-102	XDR Decoding Error
ROL_ESMSG	-103	Error sending the message
ROL_EARG	-105	Invalid arguments passed to the method
ROL_EOBJEX	-406	Object already exists
ROL_EOBJNEX	-407	Object doesn't exist
ROL_ETNEX	-408	Transaction doesn't exist
ROL_ETYNEX	-409	Type doesn't exist
ROL_ETYEX	-410	Type exists
ROL_EREFNEX	-411	Reference Type doesn't exist
ROL_EFOFF	-412	Field offset is negative

The methods of client object are very similar to that of ObjectPool object. In addition the read, write and lock operations are provided by the client since clients don't have any persistent objects.

```
TypePtr Client::defineType(kind op, int size, char *tag, FieldPtr flist,
        TypePtr ref, xdrFuncptr xdrfunc)
```

TypePtr Client::getType(kind op, int size, char *tag, FieldPtr flist,

TypePtr ref, xdrFuncptr xdrfunc, UniqueID *id)

TypePtr Client::deleteType(UniqueID *id, kind op)

UniqueID *Client::createObject(TypePtr type)

int Client::deleteObject(UniqueID *id)

int Client::read(UniqueID *id, FieldPtr field, void *value)

int Client:: read(UniqueID *id, FieldPtr field, void **value)

int Client::write(UniqueID *id, FieldPtr field, void *value, UniqueID *tid=NULL)

int Client:: write(UniqueID *id, FieldPtr field, void **value, UniqueID *tid=NULL)

int Client::fieldlock(UniqueID *id, FieldPtr field, UniqueID *tid,

bool lock_flag = TRUE)

UniqueID *Client::beginTransaction()

int Client::endTransaction(UniqueID *tid, bool commit_flag)

int Client::perror()

ROLEVENT CLASS

Constructor and Destructor

Access not given to the constructor and destructor of ROLEvent Class as ObjectPool class manages ROLEvent class objects.

Access Methods

ROLEventType ROLEvent::type()

Returns the event type(refer to ObjectPool:: setEventHandlers function

ObjectPool *ROLEvent::objpool()

Returns the objectpool the event is associated with the event.

UniqueID *ROLEvent::typeID()

Returns ID of the type if any associated with the event

UniqueID *ROLEvent::objectID()

Returns ID of the object if any associated with the event

UniqueID *ROLEvent::transID()

Returns ID of the transaction if any associated with the event

FieldPtr ROLEvent::field()

Returns the field of the object if the event is an update or lock event

int ROLEvent::returnValue(Value value)

Passes the value of the update in case of an update event, returns ROL_OK on success else ROL_ERROR

int ROLEvent::status()

Returns the status of the event

int ROLEvent::op()

Returns the operation mode in case of lock event

Value	Description
0	Request for releasing the lock
1	Request for acquiring the lock

TYPE CLASS

Constructor and Destructor

The Type object constructor does not do any initialization. The init() method does the initialization.

Initializing the type table

static int Type::typeInit()

Description: This function initializes the type table with basic types. This method must be called only once in the program and before creating any type.

Returns: ROL_OK if successful else returns a negative number.

The pointers to basic types created are *chartype, doubletype, floatype, inttype, longtype, shorttype, unsignedchar, unsignedlong, unsignedshort, unsignedint.*

The prototypes of XDR functions for basic types are:

```
bool_t xdrcharFunc(XDR *, Value, TypePtr);

bool_t xdrintFunc(XDR *, Value, TypePtr);

bool_t xdrshortFunc(XDR *, Value, TypePtr);

bool_t xdrlongFunc(XDR *, Value, TypePtr);

bool_t xdruintFunc(XDR *, Value, TypePtr);

bool_t xdrucharFunc(XDR *, Value, TypePtr);

bool_t xdrushortFunc(XDR *, Value, TypePtr);

bool_t xdrulongFunc(XDR *, Value, TypePtr);

bool_t xdrfloatFunc(XDR *, Value, TypePtr);

bool_t xdrdoubleFunc(XDR *, Value, TypePtr);
```

Initializing a type object

```
int Type::init(kind op, u_int size, char *tag, FieldPtr flist, TypePtr ref,
        xdrFuncptr xdrfunc, UniqueID *id, xdrproc_t xdrchildproc = NULL)
```

Description: This function initializes the Type object.

Arguments:

op describes various kinds of types supported by ROL.

op	value	op	value
CHAR	0	UCHAR	6
SHORT	1	USHORT	7
INT	2	UINT	8
DOUBLE	3	ULONG	9
FLOAT	4	ARRAY	10
LONG	5	STRUCT	11

Size is the size of the type, tag is the character string upto 80 characters in length used to identify different structures having same size, flist is the list of fields in a structure, ref is

the reference type in an array, xdrfunc is the XDR function for the type, id is the identification for the type and xdrchildproc is the XDR function for the reference type in case the type is an array.

Returns:

Code	Value	Description
ROL_OK	1	success
ROL_EINVOP	-301	invalid op for the type
ROL_ETEXISTS	-302	type already exists
ROL_EBKIND	-303	type is a basic kind. Basic types are created only once during initialization. Only composite types and arrays can be created later.
ROL_ESIZE	-304	size is negative
ROL_EFLIST	-305	flist argument is NULL while creating a structure type
ROL_EREF	-306	ref argument is NULL while creating an array type

Getting reference of a type given its ID

TypePtr Type::findType(kind op, UniqueID *id)

Description: This function returns the type given the object kind and its id.

Arguments: op is the kind of type(description given in the above function), id is the ID of the type.

Returns: TypePtr if type available, else NULL.

Setting a XDR function for a type

void Type::addxdrfunc(xdrFuncptr xdrfunc)

Description: This function is used to add XDR function to the type object after it has been created.

Arguments: pointer to XDR function.

Returns: none.

Member access methods

u_int Type::size()

Returns size of the type

int Type::op()

Returns kind of type

TypePtr Type::ref()

Returns the reference(or child) type in case of arrays

char *Type::tag()

Returns the tag for the type

FieldPtr Type::flist()

Returns field list (for structures only)

UniqueID *Type::typeID()

Returns type ID

xdrproc_t Type::xdrchildproc()

Returns XDR function for the reference or child type

xdrFuncptr Type::xdrfuncptr()

Returns XDR function for type object.

FIELD CLASS

Constructor and Destructor

Field::Field(char *name, TypePtr type, int offset, FieldPtr next, int *result)

Description: constructor for field object

Arguments: name is the tag for the object useful in case of creating structures, offset is the position where the field begins in the object, type is pointer to Type object the field represents, next is used in for creating field list(for structures only), result contains ROL_OK on success or ROL_ERROR if the offset is negative.

Methods used for manipulating field lists

void Field::next(FieldPtr n)

Field n is appended to this field.

FieldPtr Field::return_next()

Returns the next field object in the field list.

int Field::size()

Returns the size of field list with this pointer being the head of the list.

void Field::print()

Prints the offset and size of the Fields in the field list with this pointer being the head of the list.

Access Methods

int Field::offset()

Returns offset for the field

TypePtr Field::type()

Returns type of the field